Programmers Guide

Adaptive Server® Enterprise Extension Module for Python 15.7 SP100

Contents

Adaptive Server Enterprise Extension Module for Python

The extension module for Python, sybpydb, provides a Sybase® specific Python interface that is used to execute queries against an Adaptive Server® Enterprise database.

The extension module implements the Python Database API specification version 2.0 with extensions. For more information about the API specification, see *http://www.python.org/dev/peps/pep-0249*.

Required Components

Access to an Adaptive Server database using the Python programming language requires these components.

- sybpydb – extension module for the Python scripting language.
- Open Client SDK – provides application development tools that allow access to data source, information application or system service.

Version Requirements

Adaptive Server Enterprise Extension Module for Python has these version requirements.

- Adaptive Server Enterprise – version 15.7 or later
- Python installation – version 2.6, 2.7, or 3.1 built-in threaded mod
- Open Client SDK – version 15.7 or later

Note: For information about platform support, see the *Software Developers Kit and Open Server Installation Guide* for your platform.

Installing the Extension Module for Python

The extension module for Python is a component you can install through the Sybase Installer.

The extension module for Python is an optional installation component when you choose Custom as the installation type. The extension module is installed by default if the installation type you choose is Typical or Full. For complete installation and configuration instructions, see the *Software Developers Kit and Open Server Installation Guide* for your platform.

Extension Module for Python Configuration Overview

Complete basic configuration tasks for a Python application to make a connection and execute commands.

- Python module search path
- The name and address of the target server
- Security and Directory Services
- Runtime configuration through the `ocs.cfg` file

Python Module Search Path

Python searches for an imported module in the list of directories given by the Python variable `sys.path`.

This variable is initialized from the directory containing the application, and in the list of directories specified by the environment variable PYTHONPATH, which uses the same syntax as the shell variable PATH, that is, a list of directory names. If you have not set PYTHONPATH, or if the file is not found, the search continues in an installation-dependent default path.

To use the Adaptive Server Enterprise extension module for Python in an application, you must set either PYTHONPATH, or the Python variable `sys.path` to one of the following directory paths (these are the default directories where the different versions of the Adaptive Server Python extension module are installed):

Platform	Default Installation Path	Python Version
Windows	`%SYBASE%\%SYBASE_OCS%\python\python26_64\dll`	2.6
	`%SYBASE%\%SYBASE_OCS%\python\python27_64\dll`	2.7
	`%SYBASE%\%SYBASE_OCS%\python\python31_64\dll`	3.1
All other platforms	`$SYBASE/$SYBASE_OCS/python/python26_64r/lib`	2.6, 2.7

Platform	Default Installation Path	Python Version
	`$SYBASE/$SYBASE_OCS/python/python31_64r/lib`	3.1

Target Server Name and Address

When the Python application connects to Adaptive Server, the name of the target server is obtained from one of these sources.

1. The server name if specified in the **connect** method.
2. The DSQUERY environment variable, if the application does not specify the target server in the **connect** method.
3. The SYBASE environment variable, if DSQUERY is not set.

The target server's address is obtained from the directory service or from the platform-dependent interfaces file. Create a server entry in the interfaces file or the LDAP directory service. See the *Open Client and Open Server Configuration Guide* for your platform.

Security and Directory Services

The directory driver or security driver should be configured in the `libtcl.cfg` file in these sections.

- Directory driver in the [DIRECTORY] section.
- Security driver in the [SECURITY] section.

See the *Open Client and Open Server Configuration Guide* for your platform.

Runtime Configuration

Use the runtime configuration file `ocs.cfg` to set these values.

- Property values
- Server option values
- Server capabilities
- Debugging options

See *Using the Open Client and Open Server Runtime Configuration File* in the *Open Client Client-Library/C Reference Manual* for information about the file syntax and the properties that can be set in the file.

Developing Python Applications

Overview of how to write Python applications using the sybpydb module.

Thread Safety

Threads may share the module. However, a connection, its attributes, and any objects created using the connection cannot be shared without a locking mechanism.

A cursor is an example of an object created from a connection. To use a connection amongst multiple threads, the connection (its attributes and created objects) must be wrapped using a semaphore to implement resource locking.

Parameter Style

The module supports question mark style parameter marker formatting.

Loading the Extension Module for Python

Use the import statement to load the extension module for Python.
To use the sybpydb module from a Python script, you must first load it by importing it in the following manner:

```
import sybpydb
```

Establishing and Closing a Connection to Adaptive Server using DSN-style Connection String Properties

Use the **connect** method to open a database connection. The method supports DSN-style connection properties.

The method accepts the following keyword arguments:

- **user** – the user login name that the connection uses to log in to a server.
- **password** – the password that a connection uses when logging in to a server.
- **servername** – defines the Adaptive Server name to which client programs connect. If you do not specify **servername**, the DSQUERY environment variable defines the Adaptive Server name.
- **dsn** – the data source name. The data source name is a semicolon-separated string of name=value parts:
  - Name – a case-insensitive value that can be delimited by an equal sign (=) or semicolon (;). An attribute can have multiple synonyms. For example, **server** and **servername** refer to the same attribute.
  - Equals sign (=) – indicates the start of the value to be assigned to the Name. If there is no equals sign, the Name is assumed to be of boolean type with a value of true.

- Value – a string that is terminated by a semicolon (;). Use a backslash (\) if a semicolon or another back slash is present in the value. Values can be of type boolean, integer, or string. Valid values for Boolean types are true, false, on, off, 1, and 0.

Note: If a boolean name is present without a value, the Boolean type must be set to true.

For example:

```
sybpydb.connect(user='name', password='password string',
                              dsn=’servername=Sybase;timeout=10’)
```

Valid Attribute Names and Values

The table lists the valid attribute names and values for the **dsn** keyword argument.

Name	Description	Value
ANSINull	Determines whether evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons is ANSI-compliant. If the value is true, Adaptive Server enforces the ANSI behavior that *= NULL* and *is NULL* are not equivalent. In standard Transact-SQL, *= NULL* and *is NULL* are considered to be equivalent. This option affects *<> NULL* and *is not NULL* behavior in a similar fashion.	Boolean value. The default is false.
BulkLogin	Determines whether a connection is enabled to perform a bulk-copy operation.	Boolean value. The default is false.
ChainXacts	If true, Adaptive Server uses chained transaction behavior, that is, each server command is considered to be a distinct transaction. Adaptive Server implicitly executes a begin transaction before any of these statements: **delete**, **fetch**, **insert**, **open**, **select**, and **update**. You must still explicitly end or roll back the transaction. If false, an application must specify explicit begin transaction statements paired with commit or rollback statements.	Boolean value. The default is false.
Charset	Specifies the **charset** to be used on this connection.	String value.
Confidentiality	Whether data encryption service is performed on the connection.	Boolean value. The default is false.

Name	Description	Value
CredentialDelegation	Determines whether to allow the server to connect to a second server with the user's delegated credentials.	Boolean value. The default is false.
DetectReplay	Determines whether the connection's security mechanism detects replayed transmissions.	Boolean value. The default is false.
DetectOutOfSequence	Determines whether the connection's security mechanism detects transmissions that arrive out of sequence.	Boolean value. The default is false.
Integrity	Determines whether the connection's security mechanism performs data integrity checking.	Boolean value. The default is false.
Interfaces	The path and name of the interfaces file.	String value.
Keytab	The name and path to the file from which a connection's security mechanism reads the security key to go with the *username* value.	String value. The default is NULL, that is, the user must have established credentials before connecting.
Locale	Determines which language and character set to use for messages, datatype conversions, and datetime formats.	String value.
Language	Determines which language set to use for messages, datatype conversions, and datetime formats.	String value.
LoginTimeout	Specifies the login timeout value.	Integer value.
MaxConnect	Specifies the maximum number of simultaneously open connections that a context may have.	Integer value. Default value is 25. Negative and zero values are not allowed.
MutualAuthentication	Determines whether the server is required to authenticate itself to the client.	Boolean value. The default is false.

Name	Description	Value
NetworkAuthentication	Determines whether the connection's security mechanism performs network-based user authentication.	Boolean value. The default is false.
PacketSize	Specifies the TDS packet size.	Integer value.
Password	Specifies the password used to log in to the server.	String value.
PasswordEncryption	Determines whether the connection uses asymmetrical password encryption.	Boolean value. The default is false.
SecurityMechanism	Specifies the name of the network security mechanism that performs security services for the connection.	String value. The default value depends on security driver configuration.
Server **Servername**	Specifies the name of the server to which you are connected.	String value.
ServerPrincipalName	Specifies the network security principal name for the server to which a connection is opened.	String value. The default is NULL, which means that the connection assumes the server principal name is the same as its *ServerName* value.
Keepalive	Determines whether to use the KEEPALIVE option.	Boolean value. The default is true.
Timeout	Specifies the connection timeout value.	Integer value.
UID **User** **Username**	Specifies the name used to log in to the server.	String value.

Bulk Copy Support

Support for a bulk copy operation is an extension to the Python DBAPI. It enables you to bulk copy rows.

An example for a bulk copy in operation:

```
import syDbpydb
conn =
sybpydb.connect(dsn="user=john;bulklogin=true;chainxacts=off")
cur = conn.cursor()
cur.execute("create table mytable (id int identity, name
varchar(20))")
cur.close()

blk = conn.blkcursor()
blk.copy("mytable", direction="in", )
blk.rowxfer(["Mark"])
blk.rowxfer(["Leanne"])
blk.rowxfer(["Stanley"])
blk.done()
blk.close()

conn.close()
```

Accessing and Updating Data Using Python

After a connection is established, use a cursor object to manage the context of a fetch operation.

A cursor object provides access to methods to prepare and execute queries, and fetch rows from a result set. The cursor object is obtained from the connection object using the **cursor** method. This example shows how an application accesses and updates data:

```
import sybpydb

   #Create a connection.
   conn = sybpydb.connect(user='sa')

   # Create a cursor object.
   cur = conn.cursor()

   cur.execute("drop table footab")
   cur.execute("create table footab ( id integer, first char(20)
null, last char(50) null)")
   cur.execute("insert into footab values( ?, ?, ? )", (1, "John",
"Doe"))
   cur.execute("select * from footab")
   rows = cur.fetchall()
   for row in rows:
       print "-" * 55
       for col in range (len(row)):
          print "%s" % (row[col]),

   #Close the cursor object
   cur.close()

   #Close the connection
   conn.close()
```

Passing Input and Output Parameters to Stored Procedures

Starting with 15.7 ESD#3, the Adaptive Server Enterprise extension module for Python supports passing input and output parameters to stored procedures.

Use the **callproc()** method of the Cursor object to call a stored procedure. If there is an error in executing the stored procedure, **callproc()** throws an exception and you can retrieve the status value using the **proc_status** attribute. This support is an extension to the Python DBAPI specification.

This is a sample Python application with multiple row results:

```
import sybpydb
#Create a connection.
conn = sybpydb.connect(user='sa')
# Create a cursor object.
cur = conn.cursor()
# Call the stored procedure
try:
    cur.callproc('myproc')
    continue = True
    while(continue == True):
        row = cur.fetchall()
        continue = cur.nextset()
except sybpydb.Error:
    print("Status=%d" % cur.proc_status)
```

To specify output parameters, the extension module provides the OutParam constructor. This support is an extension to the Python DBAPI specification. The **callproc()** method returns a list of all the parameters passed to the method. If there are output parameters, and no result sets generated from the store procedure, the list contains the modified output values as soon as **callproc()** completes. However, if there are result sets, the list does not contain modified output values until all result sets from the stored procedure have been retrieved using the **fetch*()** methods and a call to **nextset()** is made to check if there are any more result sets. The **nextset()** method must be called even if only one result set is expected.

This is a sample Python application with output parameters:

```
import sybpydb
#Create a connection.
conn = sybpydb.connect(user='sa')
# Create a cursor object.
cur = conn.cursor()
cur.execute("""
    create procedure myproc
    @int1 int,
    @int2 int output
    as
    begin
```

```
        select @int2 = @int1 * @int1
    end
    """)
int_in = 300
int_out = sybpydb.OutParam(int())
vals = cur.callproc('pyproc', (int_in, int_out))
print ("Status = %d" % cur.proc_status)
print ("int = %d" % vals[1])
cur.connection.commit()
# Remove the stored procedure
cur.execute("drop procedure myproc")
cur.close()
conn.close()
```

More examples of different output parameter types are available in the sample program `callproc.py`.

Compute Rows Processing

Starting with 15.7 ESD#3, the Adaptive Server Enterprise extension module for Python supports compute rows processing.

An example is available in the sample program `compute.py`.

Parameter Support for Dynamic Statements and Stored Procedures

Starting with 15.7 ESD#4, the Adaptive Server Enterprise extension module for Python supports decimal, money, and LOB as parameters for dynamic statements and stored procedures.

Decimal and money type parameters

The following is an example usage of decimal and money as parameters for a stored procedure:

```
cur.execute("""
    create procedure pyproc
    @m1 money,
    @m2 money output,
    @d1 decimal(5,3),
    @d2 decimal(5,3) output,
    as
    begin
        select @d2 = @d1
        select @m2 = @m1
    end
    """)

dec_in = decimal.Decimal('1.23')
dec_out = sybpydb.OutParam(decimal.Decimal('0.00'))

dec_in = decimal.Decimal('1.23')
dec_out = sybpydb.OutParam(decimal.Decimal('0.00'))
m_in = decimal.Decimal('9.87')
m_out = sybpydb.OutParam(decimal.Decimal('0.00'))
```

```
vals = cur.callproc('pyproc', (int_out, dec_in, dec_out, m_in,
m_out))

print ("Status = %d" % cur.proc_status)
print ("decimal = %s" % vals[1])
print ("money= %s" % vals[3])
```

Support for date, time, datetime, and float parameters for stored procedures

The Adaptive Server Enterprise extension module for Python supports for date, time, datetime, and float parameters for stored procedures.

See `callproc.py` sample that demonstrates calling stored procedures with parameters of different datatypes including date, time, datetime, float, and integer. The sample also demonstrates the handling of output parameters.

Extension Module for Python API Reference

The sybpydb extension interface API.

Module Interface Methods

The API used for module interface.

connect

Constructs a connection object representing a connection to a database. The method accepts these keyword arguments:

- **user** – the user login name that the connection uses to log on to a server.
- **password** – the password that a connection uses when logging on to a server.
- **servername** – defines the Adaptive Server name to which client programs connects. If this argument is not specified then the DSQUERY environment variable defines the Adaptive Server name.

```
sybpydb.connect(user='name', password='password string',
servername='ase servername')
```

Module Interface Constants

The constants used for the extension module interface.

apilevel

String constant stating the supported DB API level. The default value is 2.0.

```
class sybpydb.apilevel
```

paramstyle

String constant stating the type of parameter marker formatting expected by the interface. The value for this constant is **qmark**. The interface expects question mark style parameter formatting, for example:

```
'...WHERE name=?'
```

```
class sybpydb.paramstyle
```

threadsafety

Integer constant stating the level of thread safety that the interface supports. The **threadsafety** constant value for the module is 1, which indicates that the module can be shared but not the connections.

```
class sybpydb.threadsafety
```

Connection Object Methods

The APIs used for connection objects.

close()

Closes the connection to the server. The connection is unusable after the call, and raises an exception if any operation is attempted. The same applies to cursor objects attempting to access the connection.

```
connection.close()
```

commit()

Executes the command **commit**.

```
connection.commit()
```

rollback()

Executes the command **rollback**.

```
connection.rollback()
```

cursor()

This method constructs a new cursor object using the connection.

```
connection.cursor()
```

messages()

This is a Python list object to which the module appends tuples (exception class and exception object) for all messages that the module receives for this connection. An error on any cursor obtained from the same connection object, is appended to the messages attribute of the connection object of the cursor.

```
connection.messages()
```

Usage example:

```
try:
    cur.execute("select ...")
except sybpydb.Error:
    for err in cur.connection.messages:
        print("Exception %s, Value %s", % err[0], err[1])
```

Cursor Object Methods

The API used for cursor objects.

close

```
cursor.close()
```

callproc

Calls a stored database procedure with the given name. After fetching all the result sets and rows, use the **proc_status** attribute to check the status result of the stored procedure.

```
cursor.callproc()
```

execute

Prepares and executes a query.

```
cursor.execute()
```

executemany

Prepares a database operation and executes it against all parameter sequences found in the sequence **seq_of_parameters**.

```
cursor.executemany()
```

fetch

Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

```
cursor.fetch()
```

fetchmany

Fetches the next set of rows of a query result, returning a sequence of sequences, for example, a list of tuples. An empty sequence is returned when no rows are available.

```
cursor.fetchmany()
```

fetchall

Fetches all (remaining) rows of a query result, returning them as a sequence of sequences.

```
cursor.fetchall()
```

description

A read-only attribute describing the column information.

```
cursor.description()
```

nextset

Forces the cursor to skip to the next available set, discarding any remaining rows from the current set.

```
cursor.nextset()
```

arraysize

This read/write attribute specifies the number of rows to fetch at a time with **fetchmany()**. It defaults to 1 which indicates to fetch a single row at a time.

```
cursor.arraysize()
```

proc_status

Read/write attribute specifies the number of rows to fetch at a time with **fetchmany()**. Defaults to 1 which indicates to fetch a single row at a time.

```
cursor.proc_status
```

Warning and Error Messages

All error and warning information is available through exceptions and subclasses.

Warning

Exception raised for warnings. A subclass of the Python `StandardError` exception.

```
sybpydb.Warning
```

Error

Exception that is the base class of all other exceptions defined by sybpydb. **Error** is a subclass of the Python `StandardError` exception.

```
sybpydb.Error
```

InterfaceError

Exception raised for errors that are related to the database interface rather than the database itself. It is a subclass of `Error`.

```
sybpydb.InterfaceError
```

DatabaseError

Exception raised for errors that are related to the database. It is a subclass of `Error`.

```
sybpydb.DatabaseError
```

DataError

Exception raised for errors that are related to problems with the processed data. It is a subclass of `DatabaseError`.

```
sybpydb.DataError
```

OperationalError

Exception raised for errors that are related to problems to the operation of the database but are not necessarily under the control of the programmer. It is a subclass of `DatabaseError`.

```
sybpydb.OperationalError
```

IntegrityError

Exception raised when the relational integrity of the database is affected. It is a subclass of `DatabaseError`.

```
sybpydb.IntegrityError
```

InternalError

Exception raised when the database encounters an internal error. It is a subclass of `DatabaseError`.

```
sybpydb.InternalError
```

ProgrammingError

Exception raised for programming errors. It is a subclass of `DatabaseError`

```
sybpydb.ProgrammingError
```

NotSupportedError

Exception raised when an unsupported method or database API is used. It is a subclass of `DatabaseError`.

```
sybpydb.NotSupportedError
```

BulkCursor Object Constructor

Python extension module that provides a connection object to establish a connection to the database. The connection object includes a method for creating a new `BulkCursor` object, which manages the context of a bulk operation.

The `BulkCursor` object can be constructed only from a connection object that was established with a property marking the connection for use in a bulk operation.

Usage

```
import sybpydb
conn = sybpydb.connect(dsn="user=sa;bulk=true")cur = conn.cursor()
cur.execute("create table mytable (i int, c char(10))")
blk = conn.blkcursor()
```

close()

The **close()** method of the `BulkCursor` object closes a bulk operation. Once this method has been called, the bulk cursor object cannot be used. **close()** takes no arguments.

Usage

```
import sybpydb
conn = sybpydb.connect(dsn="user=sa;bulk=true")
blk = conn.blkcursor()
bblk.close()
```

copy()

The **copy()** method of the `BulkCursor` object initializes a bulk operation.

This method accepts the following arguments:

- **tablename** – a string specifying the name of the table for the bulk operation.
- **direction** – this is a keyword argument with these values: *in* and *out*.

Usage

```
import sybpydb
conn = sybpydb.connect(dsn="user=sa;bulk=true")
cur = conn.cursor()
cur.execute("create table mytable (i int, c char(10))")
blk = conn.blkcursor()
blk.copy("mytable", direction="out")
```

done()

The **done()** method of the `BulkCursor` object marks the completion of a bulk operation. To start another operation, call the **copy()** method.

Usage

```
import sybpydb
conn = sybpydb.connect(dsn="user=sa;bulk=true")
cur = conn.cursor()
cur.execute("create table mytable (i int, c char(10)
blk = conn.blkcursor()
blk.copy("mytable", direction="in")
...
blk.done()
blk.copy("mytable", direction="out")
...
blk.done()
blk.close()
```

Glossary

Glossary of term specific to scripting languages.

- **Client-Library –** part of Open Client, a collection of routines for use in writing client applications. Client-Library is designed to accommodate cursors and other advanced features in the Sybase product line.
- **CS-Library –** included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications.
- **CT-Library –** (CT-Lib API) is part of the Open Client suite and is required to let an scripting application connect to Adaptive Server.
- **Extension or module –** the Python language can be extended by modules that are written in Python.
- **Python –** is an interpreted, general-purpose high-level programming language. For more information, go to *http://www.python.org*.
- **thread –** a path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.
- **Transact-SQL –** an enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Adaptive Server Enterprise.

Index